**Unit-II: Distributed Processing with Map-Reduce I**

**TOPICS COVERED IN THIS SESSION:**

**Hadoop I/O :**

Data Integrity - Data Integrity in HDFS - LocalFileSystem - ChecksumFileSystem - Compression - Codecs - Compression and Input Splits - Using Compression in MapReduce - Serialization - The Writable Interface - Writable Classes - Implementing a Custom Writable - Serialization Frameworks - File-Based Data Structures - SequenceFile - MapFile - Other File Formats and Column-Oriented Formats

**Map-Reduce**:

A Weather Dataset - Data Format - Analyzing the Data with Unix Tools - Analyzing the Data with Hadoop - Map and Reduce - Java MapReduce - Scaling Out - Data Flow – Combiner Functions - Running a Distributed MapReduce Job - Hadoop Streaming

**2.1. INTRODUCTION**

Hadoop comes with a set of primitives for data I/O. Some of these are techniques data integrity and compression, but deserve special consideration when dealing with multi-terabyte datasets.

**DATA INTEGRITY**

* Users of Hadoop rightly expect that no data will be lost or corrupted during storage or processing. However, because every I/O operation on the disk or network carries with it a small chance of introducing errors into the data that it is reading or writing, when the volumes of data flowing through the system are as large as the ones Hadoop is capable of handling, the chance of data corruption occurring is high.
* The usual way of detecting corrupted data is by computing a checksum for the data when it first enters the system, and again whenever it is transmitted across a channel that is unreliable and hence capable of corrupting the data. The data is deemed to be corrupt if the newly generated checksum doesn’t exactly match the original. This technique doesn’t offer any way to fix the data—it is merely error detection.
* It is possible that it’s the checksum that is corrupt, not the data, but this is very unlikely, because the checksum is much smaller than the data.
* A commonly used error-detecting code is CRC-32 (32-bit cyclic redundancy check), which computes a 32-bit integer checksum for input of any size. CRC-32 is used for checksumming in Hadoop’s ***ChecksumFileSystem***, while HDFS uses a more efficient variant called CRC-32C.
* HDFS transparently checksums all data written to it and by default verifies checksums when reading data. A separate checksum is created for every dfs.bytes-perchecksum bytes of data. The default is 512 bytes, and because a CRC-32C checksum is 4 bytes long, the storage overhead is less than 1%.
* Datanodes are responsible for verifying the data they receive before storing the data and its checksum. This applies to data that they receive from clients and from other datanodes during replication. A client writing data sends it to a pipeline of datanodes, and the last datanode in the pipeline verifies the checksum. If the datanode detects an error, the client receives a subclass of IOException, which it should handle in an application-specific manner (for example, by retrying the operation).
* When clients read data from datanodes, they verify checksums as well, comparing them with the ones stored at the datanodes. Each datanode keeps a persistent log of checksum verifications, so it knows the last time each of its blocks was verified. When a client successfully verifies a block, it tells the datanode, which updates its log. Keeping statistics such as these is valuable in detecting bad disks.
* In addition to block verification on client reads, each datanode runs a DataBlockScanner in a background thread that periodically verifies all the blocks stored on the data‐ node. This is to guard against corruption due to “***bit rot***” in the physical storage media.
* Because HDFS stores replicas of blocks, it can “heal” corrupted blocks by copying one of the good replicas to produce a new, uncorrupted replica. The way this works is that if a client detects an error when reading a block, it reports the bad block and the datanode it was trying to read from to the namenode before throwing a ChecksumException. The namenode marks the block replica as corrupt so it doesn’t direct any more clients to it or try to copy this replica to another datanode. It then schedules a copy of the block to be replicated on another datanode, so its replication factor is back at the expected level. Once this has happened, the corrupt replica is deleted.
* It is possible to disable verification of checksums by passing false to the setVerify Checksum() method on FileSystem before using the open() method to read a file. The same effect is possible from the shell by using the -ignoreCrc option with the -get or the equivalent -copyToLocal command. This feature is useful if you have a corrupt file that you want to inspect so you can decide what to do with it. For example, you might want to see whether it can be salvaged before you delete it.

**LocalFileSystem**

* The Hadoop LocalFileSystem performs client-side checksumming. This means that when you write a file called ***sample***, the filesystem client transparently creates a hidden file, ***.sample.crc***, in the same directory containing the checksums for each chunk of the file.
* The chunk size is controlled by the file.bytes-per-checksum property, which defaults to 512 bytes. The chunk size is stored as metadata in the .crc file, so the file can be read back correctly even if the setting for the chunk size has changed.
* Checksums are verified when the file is read, and if an error is detected, LocalFileSystem throws a ChecksumException.
* Checksums are fairly cheap to compute (in Java, they are implemented in native code), typically adding a few percent overhead to the time to read or write a file.
* For most applications, this is an acceptable price to pay for data integrity. It is, however, possible to disable checksums, which is typically done when the underlying filesystem supports checksums natively.
* This is accomplished by using RawLocalFileSystem in place of LocalFileSystem. To do this globally in an application, it suffices to remap the implementation for file URIs by setting the property fs.file.impl to the value org.apache.hadoop.fs.RawLocalFileSystem.
* Alternatively, you can directly create a RawLocalFileSystem instance, which may be useful if you want to disable checksum verification for only some reads.

for example:

Configuration conf = ...

FileSystem fs = **new** RawLocalFileSystem();

fs.initialize(**null**, conf);

* **ChecksumFileSystem**

LocalFileSystem uses ChecksumFileSystem to do its work, and this class makes it easy to add checksumming to other (nonchecksummed) filesystems, as Checksum FileSystem is just a wrapper around FileSystem.The general idiom is as follows:

**FileSystem rawFs = ... FileSystem**

**checksummedFs = new ChecksumFileSystem(rawFs);**

The underlying filesystem is called the raw filesystem, and may be retrieved using the getRawFileSystem() method on ChecksumFileSystem. ChecksumFileSystem has a few more useful methods for working with checksums, such as getChecksumFile() for getting the path of a checksum file for any file.

If an error is detected by ChecksumFileSystem when reading a file, it will call its reportChecksumFailure() method. The default implementation does nothing, but LocalFileSystem moves the offending file and its checksum to a side directory on the same device called bad\_files. Administrators should periodically check for these bad files and act on them.

**Compression**

File compression brings two major benefits: it reduces the space needed to store files, and it speeds up data transfer across the network or to or from disk. When dealing with large volumes of data, both savings can be significant.

There are many different compression formats, tools, and algorithms, each with different characteristics. Some of the more common ones that can be used with Hadoop are:

**1) GZIP**

* Provides High compression ratio.
* Uses high CPU resources to compress and decompress data.
* Good choice for Cold data which is infrequently accessed.
* Compressed data is not splittable and hence not suitable

for **MapReduce** jobs.

**2) BZIP2**

* Provides High compression ratio (even higher than GZIP).
* Takes long time to compress and decompress data.
* Good choice for Cold data which is infrequently accessed.
* Compressed data is ***splitable***.
* Even though the compressed data is ***splitable***, it is generally not suited for MR jobs because of high compression/decompression time.

**3) LZO**

* Provides Low compression ratio.
* Very fast in compressing and decompressing data.
* Compressed data is splitable if an appropriate indexing algorithm is used.
* Best suited for MR jobs because of property (ii) and (iii).

**4) SNAPPY**

* Provides average compression ratio.
* Aimed at very fast compression and decompression time.
* Compressed data is not splitable if used with normal file like .txt
* Generally used to compress Container file formats like Avro and

SequenceFile because the files inside a Compressed Container

file can be split.

All compression algorithms exhibit a space/time trade-off: faster compression and decompression speeds usually come at the expense of smaller space savings.

The compression typically give some control over this trade-off at compression time by offering nine different options: ***–1 means optimize for speed***, ***and -9 means optimize for space***. For example, the following command creates a compressed file file.gz using the fastest compression method:

**% gzip -1 file**

**Compression and Input Splits**

* When considering how to compress data that will be processed by MapReduce, it is important to understand whether the compression format supports splitting.
* Consider an uncompressed file stored in HDFS whose size is 1 GB. With an HDFS block size of 128 MB, the file will be stored as eight blocks, and a MapReduce job using this file as input will create eight input splits, each processed independently as input to a separate map task.
* Imagine now that the file is a gzip-compressed file whose compressed size is 1 GB. As before, HDFS will store the file as eight blocks. However, creating a split for each block will not work, because it is impossible to start reading at an arbitrary point in the gzip stream and therefore impossible for a map task to read its split independently of the others. The gzip format uses DEFLATE to store the compressed data, and DEFLATE stores data as a series of compressed blocks. The problem in this is, the block beginning and ending cannot be identified. For this reason, gzip does not support splitting.
* In this case, MapReduce will do the right thing and not try to split the gzipped file, since it knows that the input is gzip-compressed (by looking at the filename extension) and that gzip does not support splitting.
* A bzip2 file, on the other hand, does provide a synchronization marker between blocks (a 48-bit approximation of pi), so it does support splitting.

**USING COMPRESSION IN MAPREDUCE**

* If the input files are compressed, they will be decompressed automatically as they are read by MapReduce, using the filename extension to determine which codec to use.
* In order to compress the output of a MapReduce job, in the job configuration, set the mapreduce.output.fileoutputformat.compress property to true and set the mapreduce.output.fileoutputformat.compress.codec property to the classname of the compression codec you want to use.
* Alternatively, you can use the static convenience methods on FileOutputFormat to set these properties, as shown below:

**FileOutputFormat.setCompressOutput(job,true); FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);**

* The program is run over compressed input. It is not necessary to use the same compression format as the output, as follows:

% hadoop MaxTemperatureWithCompression input/ncdc/sample.txt.gz output

* Each part of the final output is compressed; in this case, there is a single part: % gunzip -c output/part-r-00000.gz

1949 111

1950 22

* If you are emitting sequence files for your output, you can set the mapreduce.out put.fileoutputformat.compress.type property to control the type of compression to use.
* The default is RECORD, which compresses individual records. Changing this to BLOCK, which compresses groups of records, is recommended because it compresses better.

**COMPRESSING MAP OUTPUT**

* Even if the MapReduce application reads and writes uncompressed data, it may benefit from compressing the intermediate output of the map phase.
* The map output is written to disk and transferred across the network to the reducer nodes, so by using a fast compressor such as LZO, LZ4, or Snappy, you can get performance gains simply because the volume of data to transfer is reduced.

**Serialization**

* *Serialization* is the process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage. *Deserialization* is the reverse process of turning a byte stream back into a series of structured objects.
* Serialization is used in two quite distinct areas of distributed data processing: for inter-process communication and for persistent storage.
* In Hadoop, inter-process communication between nodes in the system is implemented using *remote procedure calls* (RPCs). The RPC protocol uses serialization to render the message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message. In general, it is desirable that an RPC serialization format is:

***Compact***

A compact format makes the best use of network bandwidth, which is the most

scarce resource in a data center.

***Fast***

Inter-process communication forms the backbone for a distributed system, so it is

essential that there is as little performance overhead as possible for the serialization

and deserialization process.

***Extensible***

Protocols change over time to meet new requirements, so it should be

straightforward to evolve the protocol in a controlled manner for clients and

servers. For example, it should be possible to add a new argument to a method call

and have the new servers accept messages in the old format (without the new argument)

from old clients.

***Interoperable***

For some systems, it is desirable to be able to support clients that are written in

different languages to the server, so the format needs to be designed to make this

possible.

* The data format chosen for persistent storage would have different requirements from a serialization framework.
* The lifespan of an RPC is less than a second, whereas persistent data may be read years after it was written. But it turns out, the four desirable properties of an RPC’s serialization format are also crucial for a persistent storage format.
* We want the storage format to be compact (to make efficient use of storage space), fast (so the overhead in reading or writing terabytes of data is minimal), extensible (so we can transparently read data written in an older format), and interoperable (so we can read or write persistent data using different languages).
* Hadoop uses its own serialization format, Writables, which is certainly compact and fast, but not so easy to extend or use from languages other than Java.
* Because Writables are central to Hadoop (most MapReduce programs use them for their key and value types), we look at them in some depth in the next three sections, before looking at some of the other serialization frameworks supported in Hadoop. Avro (a serialization system) was designed to overcome some of the limitations of Writables.

**The Writable Interface**

The Writable interface defines two methods—one for writing its state to a DataOut

put binary stream and one for reading its state from a DataInput binary stream:

**package** org.apache.hadoop.io;

**import java.io.DataOutput**;

**import java.io.DataInput**;

**import java.io.IOException**;

**public interface Writable {**

**void write(DataOutput out) throws IOException;**

**void readFields(DataInput in) throws IOException;**

**}**

For example, let us use IntWritable. We can define use it in two ways as shown below:

**IntWritable writable = new IntWritable();**

**writable.set(163);**

Equivalently, we can use the constructor that takes the integer value:

**IntWritable writable = new IntWritable(163);**

* An example for a small helper method that wraps a java.io.ByteArrayOutputStream in a java.io.DataOutputStream (an implementation of java.io.DataOutput) to capture the bytes in the serialized stream:

**public static byte[] serialize(Writable writable) throws IOException {**

**ByteArrayOutputStream out = new ByteArrayOutputStream();**

**DataOutputStream dataOut = new DataOutputStream(out);**

**writable.write(dataOut);**

**dataOut.close();**

**return out.toByteArray();**

**}**

* An example for a helper method for deserialization

**public static byte[] deserialize(Writable writable, byte[] bytes)**

**throws IOException**

**{**

**ByteArrayInputStream in = new ByteArrayInputStream(bytes);**

**DataInputStream dataIn = new DataInputStream(in);**

**writable.readFields(dataIn);**

**dataIn.close();**

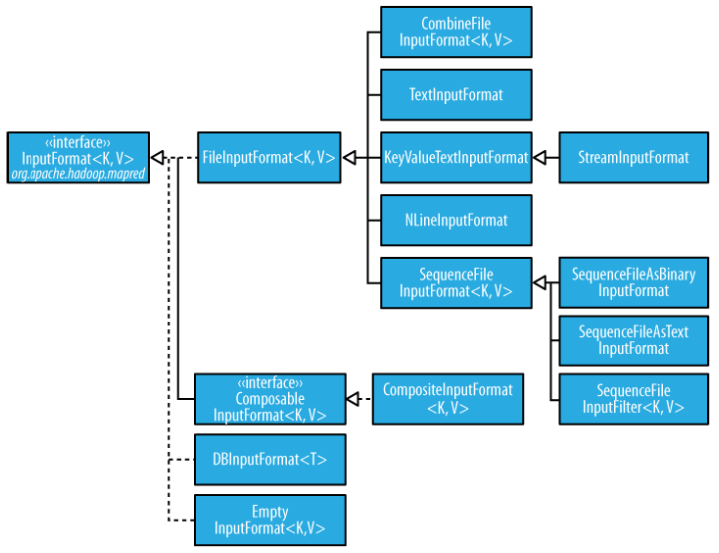
**return bytes;**

**}**

**Writable Classes**

Hadoop comes with a large selection of Writable classes, which are available in the

org.apache.hadoop.io package. They form the class hierarchy shown in



**CHOOSING BETWEEN A FIXED-LENGTH AND A VARIABLE-LENGTH ENCODING**

Fixed-length encodings are good when the distribution of values is uniform across the whole value space, such as when using a (well-designed) hash function. Most numeric variables tend to have nonuniform distributions, though, and on average, the variable-length encoding will save space. Another advantage of variable-length encodings is that you can switch from VIntWritable to VLongWritable, because their encodings are the same. So, by choosing a variable-length representation, you have room to grow without committing to an 8-byte long representation from the beginning.

**TEXT**

Text is a Writable for UTF-8 sequences. It can be thought of as the Writable equivalent of java.lang.String. But there are considerable differences between Text and String objects. They are listed below:

| PROPERTY | TEXT CLASS | STRING CLASS IN JAVA |
| --- | --- | --- |
| Native character encoding | UTF – 8 | UTF - 16 |
| Indexing | Done in terms of position in the encoded byte sequence | Unicode character in the string or the Java char code unit |
| Iteration | It is complicated using byte offsets for indexing, since you cannot just increment the index | String is not iterable, but we can iterate through |
| Mutability | Mutable | Immutable |
| Size | 2GB | 0 to 2147483647 characters |
| API and convert using toString() method | Text does not have as rich an API for manipulating strings. So they need to be converted to String using toString(). | Rich API is available for manipulating Strings. No need to convert. |

**BytesWritable**

BytesWritable is a wrapper for an array of binary data. Its serialized format is a 4-byte integer field that specifies the number of bytes to follow, followed by the bytes them‐ selves.

For example, the byte array of length 2 with values 3 and 5 is serialized as a 4- byte integer (00000002) followed by the two bytes from the array (03 and 05):

**BytesWritable b = new BytesWritable(new byte[] { 3, 5 });**

**byte[] bytes = serialize(b);**

**assertThat(StringUtils.byteToHexString(bytes), is("000000020305"));**

BytesWritable is mutable, and its value may be changed by calling its set() method.

**NullWritable**

NullWritable is a special type of Writable, as it has a zero-length serialization. No bytes are written to or read from the stream. It is used as a placeholder; for example, in Map‐ Reduce, a key or a value can be declared as a NullWritable when you don’t need to use that position, effectively storing a constant empty value. NullWritable can also be useful as a key in a SequenceFile when you want to store a list of values, as opposed to key, value pairs. It is an immutable singleton, and the instance can be retrieved by calling NullWritable.get().

**Implementing a Custom Writable**

Hadoop comes with a useful set of Writable implementations that serve most purposes; however, on occasion, you may need to write your own custom implementation. With a custom Writable, you have full control over the binary representation and the sort order. Because Writables are at the heart of the MapReduce data path, tuning the binary representation can have a significant effect on performance.

**Steps for creating custom value writable data types**

1. **Implement Writable Interface**

create custom writable should implement Writable interface to use as a map reduce jobs value class.

1. **write method**

Override write method and add logic to write all the fields value. In case of list or collections, first write size of the collection variable and then write all the value into it.

1. **readFields method**

A read fields method will read all the fields value from input stream. We must follow the same order of read and write of data members.

1. **Add Default Constructor**

The final step is to add one default constructor to allow serialization /deserialization of a custom data types.

**EXAMPLE**

**Sample Input File:**

**It consists of data in the order slno, name of the person, age and family id to which the member belongs to.**

1 John,35,1

2 Jinni,32,1

3 Ronan,3,1

4 Alex,40,2

5 Maria,36,2

6 Shira,4,2

7 Hugo,2,2

8 Erik,26,3

9 Robert,42,4

10 Anna,40,4

11 Antonio,6,4

12 Marco,4,4

13 Daniel,31,5

14 Milena,30,5

15 Brayden,2,5

16 Sergey,28,6

17 Elen,29,6

18 Eduard,32,7

19 Levon,34,8

20 Mark,30,9

### **FamilyWritable**

**package** *com.javadeveloperzone*;

**import** *org.apache.hadoop.io.IntWritable*;

**import** *org.apache.hadoop.io.Text*;

**import** *org.apache.hadoop.io.Writable*;

**import** *java.io.DataInput*;

**import** *java.io.DataOutput*;

**import** *java.io.IOException*;

**import** *java.util.ArrayList*;

**import** *java.util.List*;

**public** **class** FamilyWritable **implements** Writable

{

**private** IntWritable familyId;

**private** IntWritable totalAge;

**private** List<Text> familyMemberList;

//default constructor for (de)serialization

**public** FamilyWritable()

{

familyId = new IntWritable(0);

familyMemberList = new ArrayList<Text>();

totalAge = new IntWritable(0);

}

**public** **void** write(DataOutput dataOutput) **throws** IOException

{

familyId.write(dataOutput); //write familyId

totalAge.write(dataOutput); //write totalAge

dataOutput.writeInt(familyMemberList.size()); //write size of list

**for**(**int** index=0;index<familyMemberList.size();index++)

{

familyMemberList.get(index).write(dataOutput); //write all the value of list

}

}

**public** **void** readFields(DataInput dataInput) **throws** IOException

{

familyId.readFields(dataInput); //read familyId

totalAge.readFields(dataInput); //read totalAge

**int** size = dataInput.readInt(); //read size of list

familyMemberList = new ArrayList<Text>(size);

**for**(**int** index=0;index<size;index++)

{

//read all the values of list

Text text = new Text();

text.readFields(dataInput);

familyMemberList.add(text);

}

}

**public** IntWritable getTotalAge()

{

**return** totalAge;

}

**public** **void** setTotalAge(IntWritable totalAge)

{

**this**.totalAge = totalAge;

}

**public** IntWritable getFamilyId()

{

**return** familyId;

}

**public** **void** setFamilyId(IntWritable familyId)

{

**this**.familyId = familyId;

}

**public** List<Text> getFamilyMemberList()

{

**return** familyMemberList;

}

**public** **void** setFamilyMemberList(List<Text> familyMemberList)

{

**this**.familyMemberList = familyMemberList;

}

**public** FamilyWritable(IntWritable familyId, List<Text> familyMemberList)

{

**this**.familyId = familyId;

**this**.familyMemberList = familyMemberList;

}

**public** **void** addFamilyMember(Text familyMember,**int** age)

{

**this**.familyMemberList.add(familyMember);

**this**.totalAge.set(**this**.totalAge.get()+age);

}

**public** **void** addTotalAge(IntWritable totalAge)

{

**this**.totalAge.set(**this**.totalAge.get()+totalAge.get());

}

@Override

**public** **String** toString()

{

//average age, family member 1, family member 2... family member n

**return** (**float**)totalAge.get()/ familyMemberList.size()+","+familyMemberList.toString()

. replace("[","").replace("]","");

}

}

**Serialization Frameworks**

* Although most MapReduce programs use Writable key and value types, this isn’t mandated by the MapReduce API. In fact, any type can be used; the only requirement is a mechanism that translates to and from a binary representation of each type.
* To support this, Hadoop has an API for pluggable serialization frameworks. A serialization framework is represented by an implementation of Serialization. For example, WritableSerializationis the implementation of Serialization for Writable types.
* A Serialization defines a mapping from types to Serializer instances (for converting an object into a byte stream) and Deserializer instances (for converting a byte stream into an object).
* Set the io.serializations property to a comma-separated list of class-names in order to register Serialization implementations. Its default value includes org.apache.ha doop.io.serializer.WritableSerialization

**File-Based Data Structures**

For some applications, you need a specialized data structure to hold your data. For doing MapReduce-based processing, putting each blob of binary data into its own file does not scale, so Hadoop developed a number of higher-level containers for these situations.

***SequenceFile***:

* Imagine a logfile where each log record is a new line of text. If you want to log binary types, plain text isn’t a suitable format. Hadoop’s SequenceFile class fits this requirement, providing a persistent data structure for binary key-value pairs.
* To ***create*** a SequenceFile, use one of its createWriter() static methods, which return a SequenceFile.Writer instance.
* ***Reading*** sequence files from beginning to end is a matter of creating an instance of SequenceFile.Reader and iterating over records by repeatedly invoking one of the next() methods.
* The hadoop fs command has a -text option to display sequence files in textual form.
* The most powerful way of sorting (and merging) one or more sequence files is to use MapReduce. MapReduce is inherently parallel and will let you specify the number of reducers to use, which determines the number of output partitions.
* A sequence file consists of a header followed by one or more records. The first three bytes of a sequence file are the bytes SEQ, which act as a magic number; these are followed by a single byte representing the version number. The header contains other fields, including the names of the key and value classes, compression details, user defined metadata, and the sync marker.

***MapFile***

* A MapFile is a sorted SequenceFile with an index to permit lookups by key. The index is itself a SequenceFile that contains a fraction of the keys in the map (every 128th key, by default).
* MapFile offers a very similar interface to SequenceFile for reading and writing—the main thing to be aware of is that when writing using MapFile.Writer, map entries must be added in order, otherwise an IOException will be thrown.

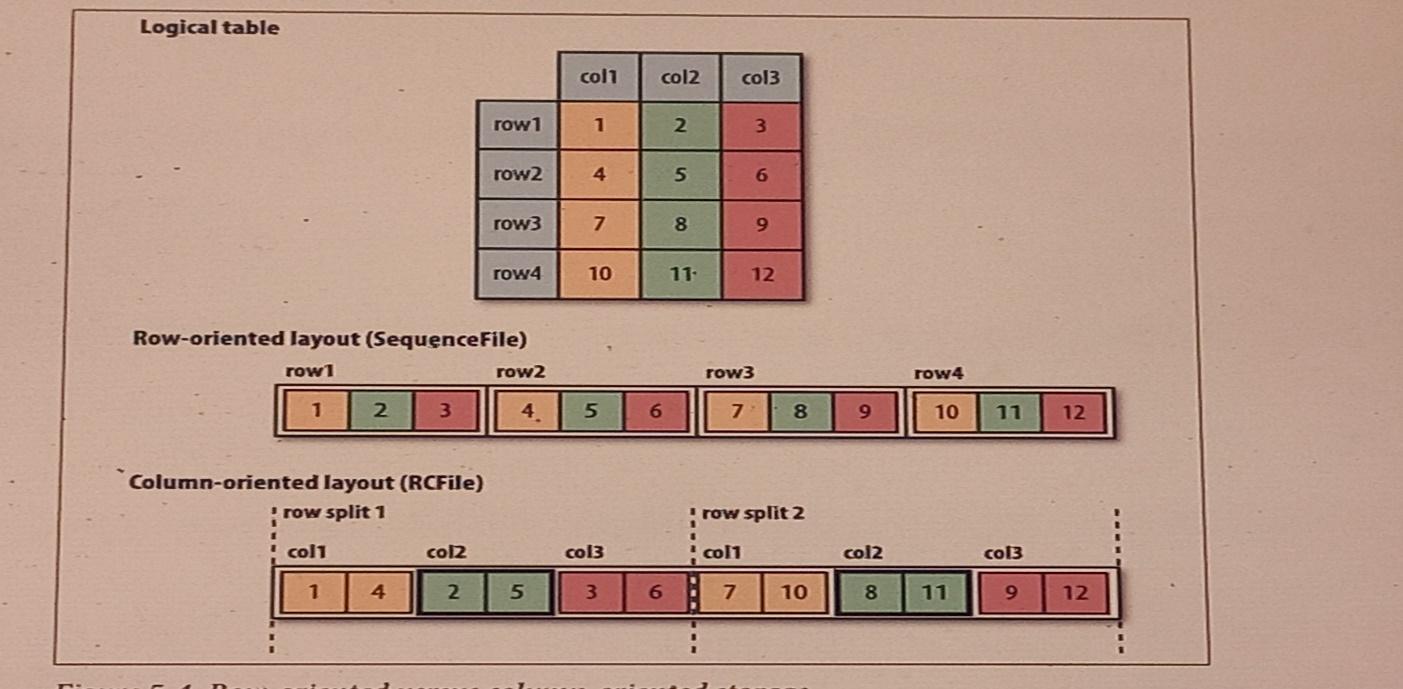
***MapFile variants***

Hadoop comes with a few variants on the general key-value MapFile interface:

* ***SetFile*** is a specialization of MapFile for storing a set of Writable keys. The keys must be added in sorted order.
* ***ArrayFile*** is a MapFile where the key is an integer representing the index of the element in the array and the value is a Writable value.
* ***BloomMapFile*** is a MapFile that offers a fast version of the get() method, especially for sparsely populated files. The implementation uses a dynamic Bloom filter for testing whether a given key is in the map.

***Column oriented files***

Sequence files, map files, and Avro datafiles are all row-oriented file formats, which means that the values for each row are stored contiguously in the file. In a column-oriented format, the rows in a file are broken up into row splits, then each split is stored in column-oriented fashion: the values for each row in the first column are stored first, followed by the values for each row in the second column, and so on. This is shown diagrammatically in following figure.



* Column-oriented formats need more memory for reading and writing, since they have to buffer a row split in memory, rather than just a single row.
* Column-oriented formats are not suited to streaming writes, as the current file cannot be recovered if the writer process fails.
* On the other hand, row-oriented formats like sequence files and Avro datafiles can be read up to the last sync point after a writer failure. It is for this reason that Flume uses row-oriented formats.
* The first column-oriented file format in Hadoop was Hive’s ***RCFile***, short for Record Columnar File. It has been superseded by Hive’s ***ORCFile*** (Optimized Record Columnar File), and Parquet.
* ***Parquet*** is a general-purpose column-oriented file format based on Google’s ***Dremel***, and has wide support across Hadoop components. Avro also has a column-oriented format called ***Trevni***.

MapReduce is a programming model for data processing. The model is simple, yet not too simple to express useful programs in. Hadoop can run MapReduce programs written in various languages.

**A WEATHER DATASET**

For our example, we will write a program that mines weather data. Weather sensors

collect data every hour at many locations across the globe and gather a large volume of log data, which is a good candidate for analysis with MapReduce because we want to process all the data, and the data is semi-structured and record-oriented.

**DATA FORMAT**

The data we will use is from the National Climatic Data Center, or NCDC. The data is

stored using a line-oriented ASCII format, in which each line is a record. The format

supports a rich set of meteorological elements, many of which are optional or with

variable data lengths. For simplicity, we focus on the basic elements, such as temperature, which are always present and are of fixed width.

A sample line with some of the salient fields annotated. The line has been split into multiple lines to show each field; in the real file, fields are packed into one line with no delimiters.

**0057**

**332130 # USAF weather station identifier**

**99999 # WBAN weather station identifier**

**19500101 # observation date**

**0300 # observation time**

**4**

**+51317 # latitude (degrees x 1000)**

**+028783 # longitude (degrees x 1000)**

**FM-12**

**+0171 # elevation (meters)**

**99999**

**V020**

**320 # wind direction (degrees)**

**1 # quality code**

**N**

**0072**

**1**

**00450 # sky ceiling height (meters)**

**1 # quality code**

**C**

**N**

**010000 # visibility distance (meters)**

**1 # quality code**

**N**

**9**

**-0128 # air temperature (degrees Celsius x 10)**

**1 # quality code**

**-0139 # dew point temperature (degrees Celsius x 10)**

**1 # quality code**

**10268 # atmospheric pressure (hectopascals x 10)**

**1 # quality code**

Datafiles are organized by date and weather station. There is a directory for each year from 1901 to 2001, each containing a gzipped file for each weather station with its readings for that year. For example, here are the first entries for 1990:

% **ls raw/1990 | head**

**010010-99999-1990.gz**

**010014-99999-1990.gz**

**010015-99999-1990.gz**

**010016-99999-1990.gz**

**010017-99999-1990.gz**

**010030-99999-1990.gz**

**010040-99999-1990.gz**

**010080-99999-1990.gz**

**010100-99999-1990.gz**

**010150-99999-1990.gz**

There are tens of thousands of weather stations, so the whole dataset is made up of a large number of relatively small files. It’s generally easier and more efficient to process a smaller number of relatively large files, so the data was pre-processed so that each year’s readings were concatenated into a single file.

**ANALYSING THE DATA WITH UNIX TOOLS**

***What is the highest recorded global temperature for each year in the dataset?***

* The first solution is provided without using Hadoop which will provide a performance baseline and a useful means to check our results.

The classic tool for processing line-oriented data is ***awk***

*#!/usr/bin/env bash*

**for** year in all /\*

**do**

**echo -ne `basename $year .gz`"\t"**

**gunzip -c $year | \**

**awk '{ temp = substr($0, 88, 5) + 0;**

**q = substr($0, 93, 1);**

**if (temp !=9999 && q ~ /[01459]/ && temp > max) max = temp }**

**END { print max }'**

**done**

* The script loops through the compressed year files, first printing the year, and then processing each file using ***awk***.
* The *awk* script extracts two fields from the data: the air temperature and the quality code.
* The air temperature value is turned into an integer by adding 0. Next, a test is applied to see whether the temperature is valid (***the value 9999 signifies a missing value in the NCDC dataset***) and whether the quality code indicates that the reading is not suspect or erroneous.
* If the reading is OK, the value is compared with the maximum value seen so far, which is updated if a new maximum is found.
* The END block is executed after all the lines in the file have been processed, and it prints the maximum value.

Here is the beginning of a run:

% **./max\_temperature.sh**

1901 317

1902 244

1903 289

1904 256

1905 283

**...**

* The temperature values in the source file are scaled by a factor of 10, so this works out as a maximum temperature of 31.7°C for 1901.
* The complete run for the century took ***42 minutes*** in one run on a single EC2 High-CPU Extra Large instance.

**Analysing the Data with Hadoop**

To take advantage of the parallel processing that Hadoop provides, we need to express our query as a MapReduce job.

**Map and Reduce**

* MapReduce works by breaking the processing into two phases: ***the map phase*** and ***the reduce phase***.
* Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer.
* The programmer also specifies two functions: ***the map function*** and ***the reduce function.***
  + The input to our map phase is the raw NCDC data.
  + The map function is simple. We extract the year and the air temperature, because these are the only fields of interest.
  + Our map function is simple. We pull out the year and the air temperature, because these are the only fields we are interested in. In this case, the map function is just a data preparation phase, setting up the data in such a way that the reduce function can do its work on it: finding the maximum temperature for each year. The map function is also a good place to drop bad records: here we filter out temperatures that are missing, suspect, or erroneous.
  + The output from the map function is processed by the MapReduce framework before being sent to the reduce function. This processing sorts and groups the key-value pairs by key. So, continuing the example, our reduce function sees the following input:
  + Each year appears with a list of all its air temperature readings. What the reduce function has to do now, is iterate through the list and pick up the maximum reading:

(1949, 111)

(1950, 22)

This is the final output: the maximum global temperature recorded in each year.

**SCALING OUT**

Using smaller data sets, it is easy to verify that MapReduce works for small inputs; and they can work using the files that are on local file system. But if the data size is huge, we need to store the data in a distributed filesystem (typically HDFS). This allows Hadoop to move the MapReduce computation to each machine hosting a part of the data, using Hadoop’s resource management system, called YARN.

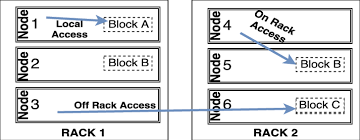
**DATA FLOW(actual content starts at the line marked (\*\*) )**

To learn about data flow, first the terminology associated with it is described below:

* **JOB:** A MapReduce ***job*** is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information.
* **TASK:** Hadoop runs the job by dividing it into ***tasks***. The tasks are of two types: ***map tasks*** and ***reduce tasks***. The tasks are scheduled using YARN and run-on nodes in the cluster. If a task fails, it will be automatically rescheduled to run on a different node.
* **SPLIT:** Hadoop divides the input to a MapReduce job into fixed-size pieces called ***input splits***, or just ***splits***. Hadoop creates one map task for each split, which runs the user-defined map function for each record in the split.
  + - Having many splits means the time taken to process each split is small compared to the time to process the whole input. So, if we are processing the splits in parallel, the processing is better load balanced when the splits are small, since a faster machine will be able to process proportionally more splits over the course of the job than a slower machine. Even if the machines are identical, failed processes or other jobs running concurrently make load balancing desirable, and the quality of the load balancing in‐ creases as the splits become finer grained.
    - On the other hand, if splits are too small, the overhead of managing the splits and map task creation begins to dominate the total job execution time. For most jobs, a good split size tends to be the size of an HDFS block, which is ***128 MB*** by default, although this can be changed for the cluster (for all newly created files) or specified when each file is created.

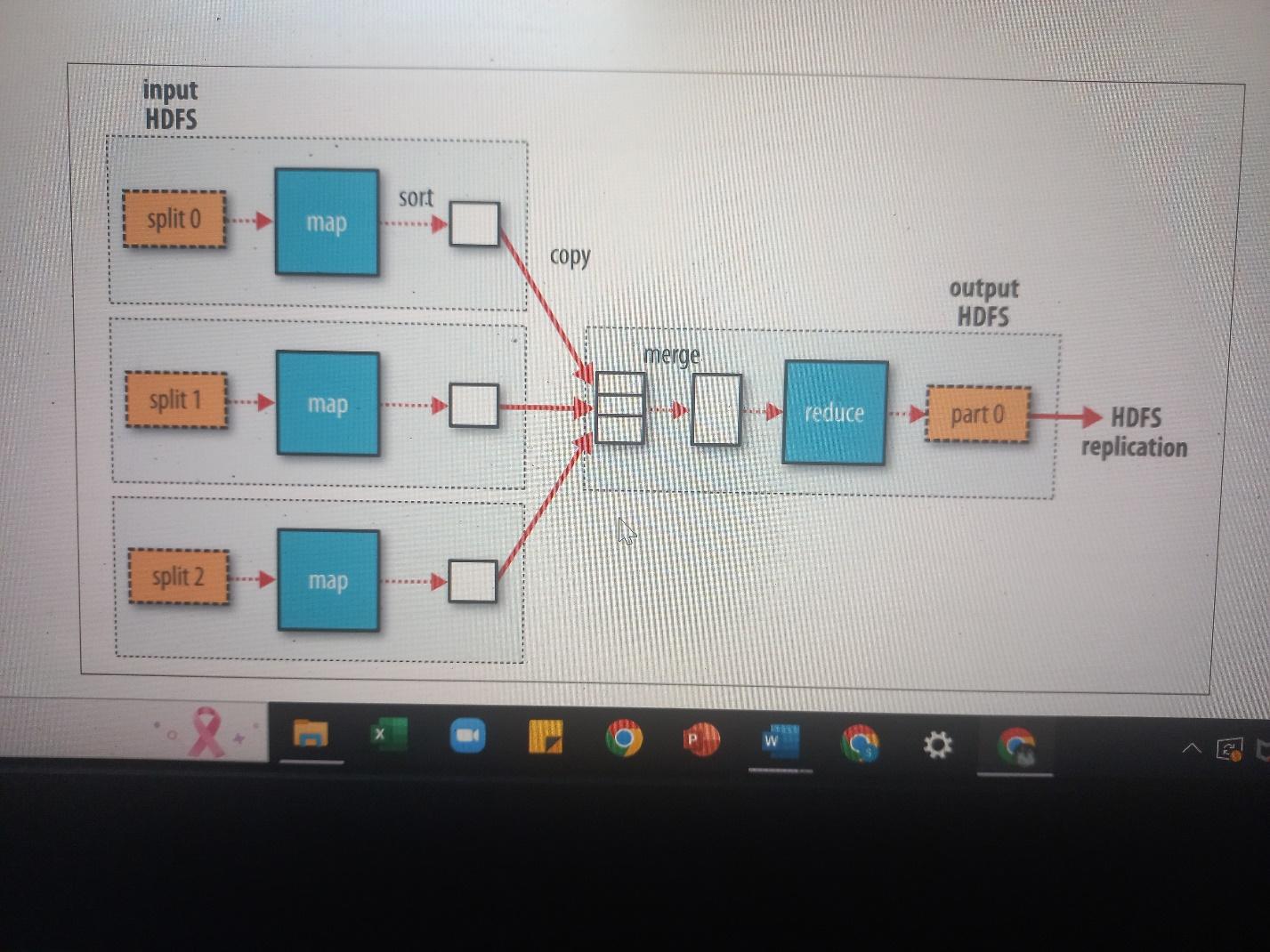
(\*\*)

* Hadoop does its best to run the map task on a node where the input data resides in HDFS, because it doesn’t use valuable cluster bandwidth. This is called data locality optimization. Sometimes, however, all the nodes hosting the HDFS block replicas for a map task’s input split are running other map tasks, so the job scheduler will look for a free map slot on a node in the same rack as one of the blocks. Very occasionally even this is not possible, so an off-rack node is used, which results in an inter-rack network transfer. The three possibilities are illustrated in the following Figure.



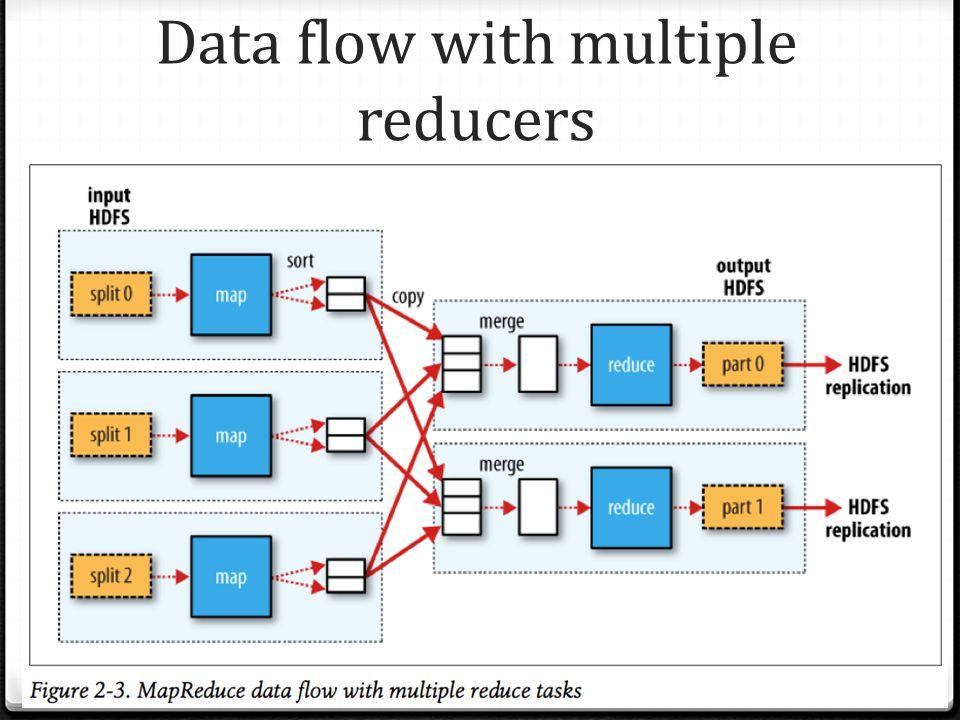
***Figure: Data-local (a), rack-local (b), and off-rack (c) map tasks***

* It should now be clear why the optimal split size is the same as the block size: it is the largest size of input that can be guaranteed to be stored on a single node. If the split spanned two blocks, it would be unlikely that any HDFS node stored both blocks, so some of the split would have to be transferred across the network to the node running the map task, which is clearly less efficient than running the whole map task using local data.
* Map tasks write their output to the local disk, not to HDFS. Because, Map output is intermediate output: it’s processed by reduce tasks to produce the final output, and once the job is complete, the map output can be thrown away. So, storing it in HDFS with replication would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to re-create the map output.
* Reduce tasks don’t have the advantage of data locality; the input to a single reduce task is normally the output from all mappers. In the present example, we have a single reduce task that is fed by all of the map tasks. Therefore, the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function. The output of the reduce is normally stored in HDFS for reliability.
* For each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes for reliability. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes.
* The whole data flow with a single reduce task is illustrated in the following figure. The dotted boxes indicate nodes, the dotted arrows show data transfers on a node, and the solid arrows show data transfers between nodes.

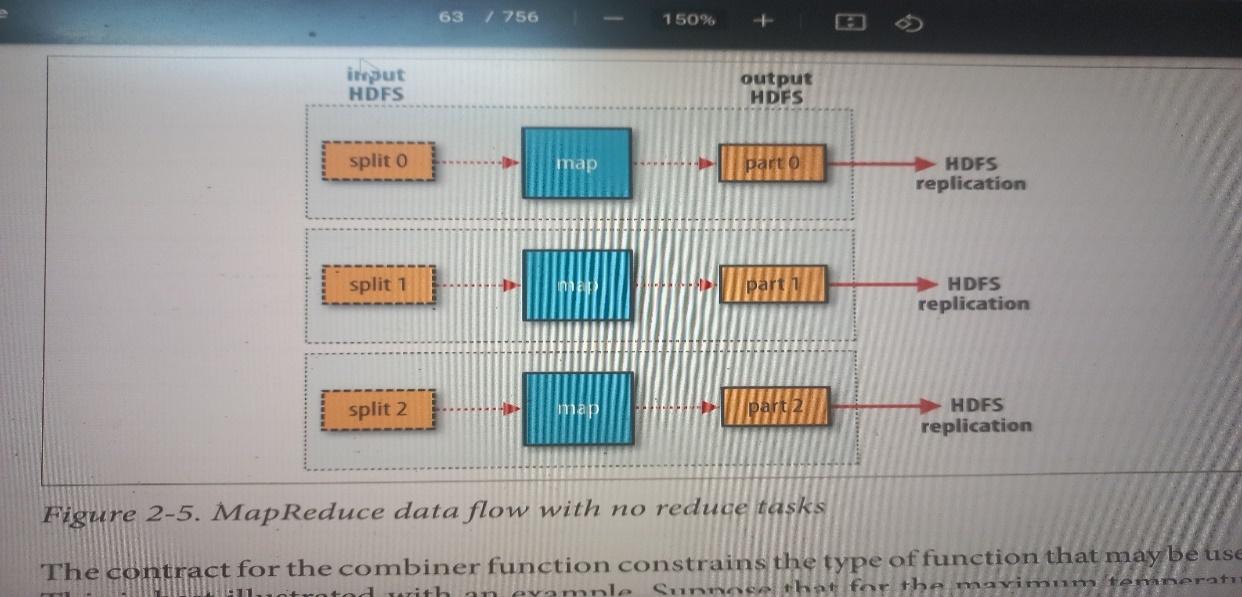


***Figure: MapReduce data flow with a single reduce task***

* The number of reduce tasks is not governed by the size of the input, but instead is specified independently.
* When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.
* The data flow for the general case of multiple reduce tasks is illustrated in the following figure. This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as “the shuffle,” as each reduce task is fed by many map tasks. The shuffle is more complicated than this diagram suggests, and tuning it can have a big impact on job execution time.



* It’s also possible to have zero reduce tasks. This can be appropriate when we don’t need the shuffle because the processing can be carried out entirely in parallel. In this case, the map tasks write the output to HDFS. This is shown in the following figures:



***Figure: Data flow with zero reducers***

**RUNNING A DISTRIBUTED MAPREDUCE JOB**

The same program will run, without alteration, on a full dataset. MapReduce scales to the size of a data and the size of your hardware. On a 10-node EC2 cluster running High-CPU Extra Large instances, the program took ***six minutes*** to run.

**HADOOP STREAMING**

* Hadoop provides an API to MapReduce that allows users to write map and reduce functions in languages other than Java.
* Hadoop Streaming uses Unix standard streams as the interface between Hadoop and the program, so users can use any language that can read standard input and write to standard output to write your MapReduce program.

Streaming is naturally suited for text processing. Map input data is passed over standard input to your map function, which processes it line by line and writes lines to standard output. A map output key-value pair is written as a single tab-delimited line. Input to the reduce function is in the same format—a tab-separated key-value pair—passed over standard input. The reduce function reads lines from standard input, which the frame‐ work guarantees are sorted by key, and writes its results to standard output.